6 OAuth 2.0 in the real world

This chapter covers

- Using different OAuth grant types for different situations
- Dealing with native web and browser-based applications
- Handling secrets at configuration time and runtime

So far in this book, we've covered OAuth 2.0 in a fairly idealized state. All the applications look the same, all the resources look the same, and everybody does things the same way. Our extensive example in chapter 2 covered the authorization grant protocol, using a web application with a client secret. All of our exercises in chapters 3, 4, and 5 have made use of the same setup.

Making simplifying assumptions like this is a good way to learn about the fundamentals of a system, but of course the applications we all build in the real world need to live in the real world with all of its variations. OAuth 2.0 anticipates these variations in several ways by allowing flexibility in the OAuth protocol in key places. In this chapter, we'll look at some of those extension points in greater detail.

6.1 Authorization grant types

In OAuth 1.0, there was only one method for getting an access token that all clients had to use. It was designed to be as general purpose as possible, attempting to cater to a wide variety of deployment options. As a consequence, the protocol was not particularly well suited to any use case. Web applications needed to deal with

request tokens that were intended for native applications to poll for state changes, native applications needed to deal with consumer secrets that were intended to protect web applications, and everybody had to deal with a customized signature mechanism. It worked well enough to cement OAuth as a powerful and foundational technology, but it left a lot to be desired.

When OAuth 2.0 was being developed, the working group made a distinct decision to treat the core protocol as a *framework* instead of a single protocol. By keeping the core concepts of the protocol solid and allowing extensions in specific areas, OAuth 2.0 can be applied in many different ways. Although it has been argued that the second version of any system will turn into an abstract framework,[1] in OAuth's case the abstractions have helped immensely in extending its applicability and usefulness.

One of the key areas that OAuth 2.0 can vary is that of the *authorization grant*, colloquially known as the *OAuth flow*. As we hinted at in previous chapters, the authorization code grant type is just one of several different ways that an OAuth client can get a token from an authorization server. Since we've covered the authorization code grant in great detail already, we'll be looking at the other primary options here in this section.

6.1.1 Implicit grant type

One key aspect of the different steps in the authorization code flow is that it keeps information separate between different components. This way, the browser doesn't learn things that only the client should know about, and the client doesn't get to see the state of the browser, and so on. But what if we were to put the client *inside* the browser (see figure 6.1)?

This is what happens with a JavaScript application running completely inside the browser. The client then can't keep any secrets from the browser, which has full insight into the client's execution. In this case, there is no real benefit in passing the authorization code through the browser to the client, only to have the client exchange that for a token because the extra layer of secrets isn't protected against anyone involved.

The *implicit grant type* does away with this extra secret and its attendant round trip by returning the token directly from the authorization endpoint. The implicit grant type therefore uses only the front channel[2] to communicate with the authorization server. This flow is very useful for JavaScript applications embedded within websites that need to be able to perform an authorized, and potentially limited, session sharing across security domains.

The implicit grant has severe limitations that need to be considered when approaching it. First, there is no realistic way for a client using this flow to keep a client secret, since the secret will be made available to the browser itself. Since this flow uses only the authorization endpoint and not the token endpoint, this limitation does not affect its

[1] This is known as "second system syndrome" and has been well studied. This syndrome has been known to kill perfectly reasonable solutions with too much abstraction and complexity. That's probably not what will happen with OAuth 2.0, though. We hope.

[2] We covered the front channel and back channel in chapter 2, remember?

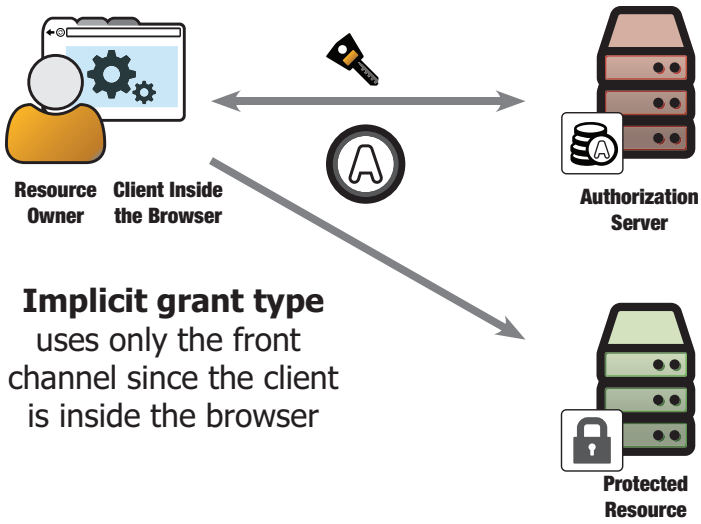

Figure 6.1 The implicit grant type

ability to function, as the client is never expected to authenticate at the authorization endpoint. However, the lack of any means of authenticating the client does impact the security profile of the grant type and it should be approached with caution. Additionally, the implicit flow can't be used to get a refresh token. Since in-browser applications are by nature short lived, lasting only the session length of the browser context that has loaded them, the usefulness of a refresh token would be very limited. Furthermore, unlike other grant types, the resource owner can be assumed to be still present in the browser and available to reauthorize the client if necessary. Authorization servers are still able to apply Trust On First Use (TOFU) principles, allowing a reauthentication to be a seamless user experience.

The client sends its request to the authorization server's authorization endpoint in the same manner as the authorization code flow, except that this time the `response_type` parameter is set to `token` instead of `code`. This signals the authorization server to generate a token immediately instead of generating an authorization code to be traded in for a token.

```
HTTP/1.1 302 Moved Temporarily
Location: http://localhost:9001/authorize?response_type=token&scope=foo&client_
id=oauth-client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&state
=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 444
Date: Fri, 31 Jul 2015 20:50:19 GMT
```

The client can do this by using a full-page redirect or by using an inline frame (iframe) inside the page itself. Either way, the browser makes a request to the authorization server's authorization endpoint. The resource owner authenticates themselves and

authorizes the client in the same manner as the authorization code flow. However, this time the authorization server generates the token immediately and returns it by attaching it to the URI fragment of the response from the authorization endpoint. Remember, since this is the front channel, the response to the client comes in the form of an HTTP redirect back to the client's redirect URI.

```
GET /callback#access_token=987tghjkiu6trfghjuytrghj&token_type=Bearer
HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:9001/authorize?response_type=code&scope=foo&client_
id=oauth-client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&state
=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1
```

The fragment portion of the URI isn't usually sent back to the server, which means that the token value itself is available only inside the browser. Note, however, that this behavior does vary depending on the browser implementation and version.

Let's try our hand at implementing this. Open up `ch-6-ex-1` and edit the `authorizationServer.js` file to begin. Inside the function that handles the submission from our approval page, we've already got a branch of an `if` statement that deals with the `response_type` of `code`.

```
if (query.response_type == 'code') {
```

We're going to add a branch to this block that will handle the `response_type` of `token` instead.

```
} else if (query.response_type == 'token') {
```

Inside this new block, we're going to need to do the same kinds of processing on the request as we did for the authorization code grant, checking scopes and verifying the approval against the request. Note that the error is sent back as the hash and not as the query parameter.

```
var rscope = getScopesFromForm(req.body);
var client = getClient(query.client_id);
var cscope = client.scope ? client.scope.split(' ') : undefined;
if (__.difference(rscope, cscope).length > 0) {
  var urlParsed = buildUrl(query.redirect_uri,
       {},
       qs.stringify({error: 'invalid_scope'})
  );
  res.redirect(urlParsed);
  return;
}
```

We'll then generate the access token as we normally would. Remember that we don't create a refresh token.

```
var access_token = randomstring.generate();
nosql.insert({ access_token: access_token, client_id: clientId, scope: rscope });
```

```
var token_response = { access_token: access_token, token_type: 'Bearer',
scope: rscope.join(' ') };
if (query.state) {
       token_response.state = query.state;
}
```

Finally, send this back to the client using the hash fragment of the redirect URI.

```
var urlParsed = buildUrl(query.redirect_uri,
   {},
   qs.stringify(token_response)
);
res.redirect(urlParsed);
return;
```

We'll take a look at the details of the implementation of the client side in section 6.2.2 when we cover in-browser clients. For now, you should be able to load up the client page at `http://localhost:9000/` and have the client get an access token and call the protected resource as you'd expect to in other exercises. When you return from the authorization server, notice that your client comes back with the token value itself in the hash of the redirect URI. The protected resource doesn't need to do anything different to process and validate this token, but it does need to be configured with cross-origin resource sharing (CORS), which we'll cover in chapter 8.

6.1.2 Client credentials grant type

What if there is no explicit resource owner, or the resource owner is indistinguishable from the client software itself? This is a fairly common situation, in which there are back-end systems that need to communicate directly with each other and not necessarily on behalf of any one particular user. With no user to delegate the authorization to the client, can we even do OAuth (see figure 6.2)?

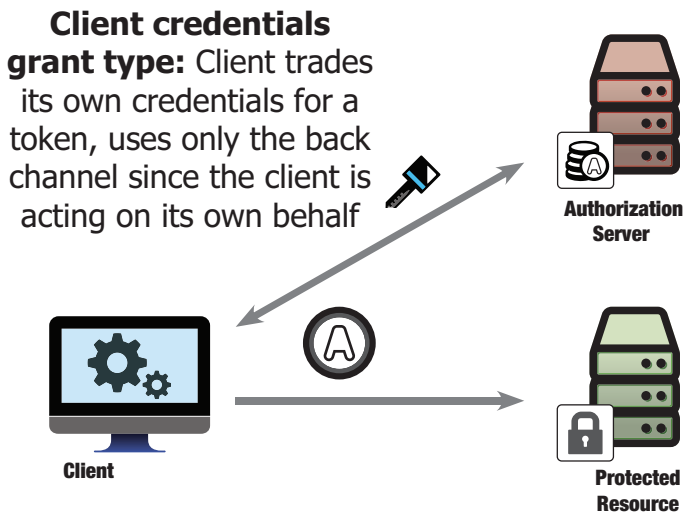

Figure 6.2 The client credentials grant type

We can, by making use of the client credentials grant type that was added to OAuth 2.0 for this case. In the implicit flow, the client is pushed up into the browser and therefore into the front channel; in this flow, the resource owner is pushed down into the client and the user agent disappears from the picture. As a consequence, this flow makes exclusive use of the back channel, and the client acts on its own behalf (as its own resource owner) to get the access token from the token endpoint.

OAuth's legs

In OAuth 1.0, there was no mechanism for a client getting its own token as the protocol was designed around allowing users to delegate access, a "three-legged" protocol between the client, server, and user. However, people deploying OAuth 1.0 realized fairly quickly that it was useful to use some of OAuth's mechanisms to connect back-end services in lieu of API keys. This was dubbed "two-legged OAuth" because it no longer involved the resource owner, only the client and the resource server. But instead of using OAuth's tokens, people decided to use OAuth 1.0's signature mechanism alone to create a signed request from the client to the resource server. This required the resource server to know the client's secret in order to validate the signature of the request. Since there is no token or credential exchange occurring, it could more properly be termed "zero-legged OAuth."

When OAuth 2.0 was being designed, the working group looked at the deployment patterns of OAuth 1.0 and decided to codify the pattern of a client accessing a protected resource on its own behalf, but this time it would use as much of the same token mechanisms that the three-legged delegation flows used. This parallelism ensures that the authorization server is still the component in charge of the client's credentials, allowing the resource server to deal with tokens alone. Whether the tokens are delegated from end users or given directly to clients, the resource server can handle them in the same fashion, simplifying the code base and architectural considerations across the entire OAuth system.

The client requests a token from the token endpoint as it would with the authorization code grant, except that this time it uses the `client_credentials` value for the `grant_type` parameter and doesn't have an authorization code or other temporary credential to trade for the token. Instead, the client authenticates itself directly, and the authorization server issues an appropriate access token. The client can also request specific scopes inside this call using the `scope` parameter, analogous to the `scope` parameter used at the authorization endpoint by the authorization code and implicit flows.

```
POST /token
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3JldC0x

grant_type=client_credentials&scope=foo%20bar
```

The response from the authorization server is a normal OAuth token endpoint response: a JSON object containing the token information. The client credentials flow does not issue a refresh token because the client is assumed to be in the position of being able to request a new token for itself at any time without involving a separate resource owner, which renders the refresh token unnecessary in this context.

```
HTTP 200 OK
Date: Fri, 31 Jul 2015 21:19:03 GMT
Content-type: application/json

{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "scope": "foo bar",
   "token_type": "Bearer"
}
```

The client uses this access token in the same manner that it would an access token gathered through a different flow, and the protected resource doesn't even necessarily have to know how the token was acquired. The tokens themselves will likely have different rights of access associated with them, depending on whether they were delegated by a user or requested directly by a client, but that kind of differentiation can be handled with an authorization policy engine that can tell those cases apart. In other words, even though the tokens look the same coming through the front door of the resource server, they can still mean different things.

Let's build this functionality into our server and client. Open up `ch-6-ex-2` and edit the `authorizationServer.js` file. We're going to go into the handler for the token endpoint and find the section of code that handles the token requests for the authorization code grant type.

```
if (req.body.grant_type == 'authorization_code') {
```

We're going to add a branch to this `if` statement to handle the client credentials grant type.

```
} else if (req.body.grant_type == 'client_credentials') {
```

At this point, our code has already verified the client's ID and secret that was presented to the token endpoint, and we now need to figure out whether the request coming in can be made into a token for this particular client. We can perform a number of checks here, including checking whether the scopes requested match what the client is allowed to ask for, checking whether the client is allowed to use this grant type, or even checking to see whether or not this client already has an access token in transit that we might want to revoke preemptively. In our simple exercise, we'll check the scopes here, and we'll borrow the scope matching code from the authorization code grant type to do that.

```
var rscope = req.body.scope ? req.body.scope.split(' ') : undefined;
var cscope = client.scope ? client.scope.split(' ') : undefined;
if (__.difference(rscope, cscope).length > 0) {
   res.status(400).json({error: 'invalid_scope'});
   return;
}
```

Scopes and grant types

Since the client credentials grant type doesn't have any direct user interaction, it's really meant for trusted back-end systems accessing services directly. With that kind of power, it's often a good idea for protected resources to be able to differentiate between interactive and noninteractive clients when fulfilling requests. A common method of doing this is to use different scopes for both classes of clients, managing them as part of the client's registration with the authorization server.

With that sorted out, we can now issue the access token itself. We'll save it to our database, as we did previously.

```
var access_token = randomstring.generate();
var token_response = { access_token: access_token, token_type: 'Bearer',
scope: rscope.join(' ') };
nosql.insert({ access_token: access_token, client_id: clientId, scope:
rscope });
res.status(200).json(token_response);
return;
```

Now we'll turn our attention to the client. Edit `client.js` in the same exercise and find the function that handles authorizing the client.

```
app.get('/authorize', function(req, res){
```

Instead of redirecting the resource owner, this time we're going to call the token endpoint directly. We'll base this off the code we used for handling the callback URI in the authorization code grant. It's a simple HTTP POST, and we're going to include our client credentials as HTTP Basic authentication.

```
var form_data = qs.stringify({
   grant_type: 'client_credentials',
   scope: client.scope
});
var headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'Authorization': 'Basic ' + encodeClientCredentials(client.client_id,
   client.client_secret)
};

var tokRes = request('POST', authServer.tokenEndpoint, {
   body: form_data,
   headers: headers
});
```

We then parse the token response as we did previously, except that this time we don't have to worry about a refresh token. Why not? Since the client can easily request a new token on its own behalf at any time without user intervention, there's no need to ever specify a refresh token in this case.

```
if (tokRes.statusCode >= 200 && tokRes.statusCode < 300) {
   var body = JSON.parse(tokRes.getBody());
```

```
    access_token = body.access_token;

    scope = body.scope;

    res.render('index', {access_token: access_token, scope: scope});
} else {
    res.render('error', {error: 'Unable to fetch access token, server
    response: ' + tokRes.statusCode})
}
```

From here, the client can call the resource server as it did previously. The protected resource doesn't have to change any of its processing code because it's receiving and validating an access token.

6.1.3 Resource owner credentials grant type

If the resource owner has a plain username and password at the authorization server, then it could be possible for the client to prompt the user for these credentials and trade them for an access token. The resource owner credentials grant type, also known as the password flow, allows a client to do just that. The resource owner interacts directly with the client and never with the authorization server itself. The grant type uses the token endpoint exclusively, remaining confined to the back channel (figure 6.3).

This method should sound eerily familiar to you at this point. "Wait a minute," you may be thinking, "we covered this back in chapter 1 and you said it was a bad

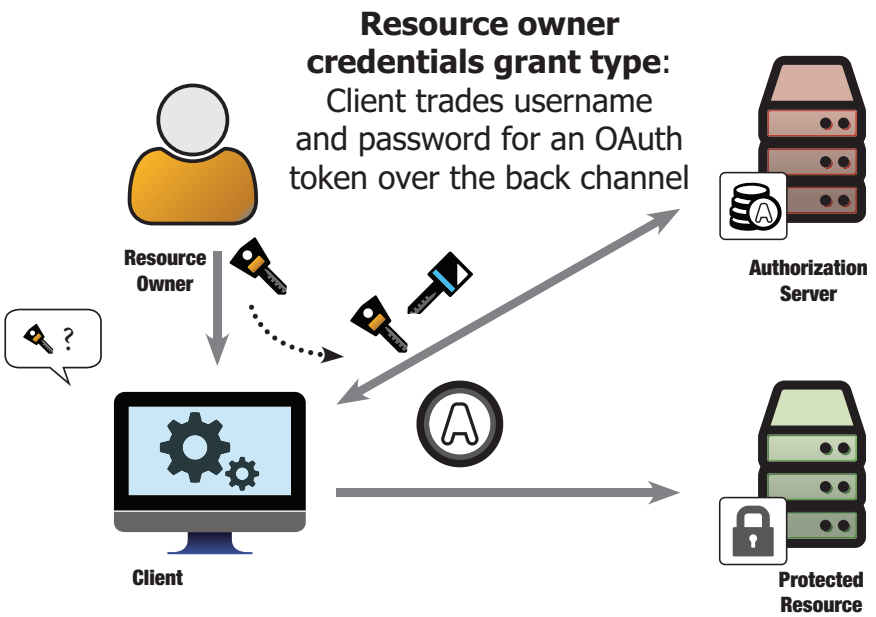

Figure 6.3 The resource owner credentials grant type

idea!" And you'd be correct: this grant type, which is included in the core OAuth specification, is based on the "ask for the keys" antipattern. And, in general, it's a bad idea.

Codifying the antipattern

Let's review: why shouldn't you use this pattern? It's certainly simpler to program than dealing with all of the back-and-forth redirects. But with that simplicity comes significantly increased security risk and decreased flexibility and functionality. The resource owner's credentials are exposed in plain text to the client, which could cache them or replay them whenever it sees fit. The credentials are presented in plain text (though over a TLS encrypted connection) to the authorization server, which then needs to verify them, leaving another potential attack surface. Unlike OAuth tokens, which can be revoked and rotated without impact to the user experience, a user's username and password tend to be much more difficult to manage and change. The requirement to collect and replay the user's credentials also limits the kinds of credentials that can be used to authenticate the user. Although an authorization server accessed through a web browser can employ a wide variety of primary authentication technologies and user experiences, such as certificates or identity federation, many of the most effective and secure ones are designed to prevent the kind of credential replay that this grant type depends on. This effectively limits the authentication to a plain username and password or its analogs. Finally, this approach trains users to give their password to any application that asks for it. Instead of this, we should be training users to give their passwords only to a core set of trusted applications, such as the authorization server.

Why, then, would OAuth codify such bad practice? When there are any other options available, this grant type is a pretty bad idea, but there aren't always other viable options. This grant type is intended for clients that would normally be prompting for the resource owner's username and password anyway, and then replaying those credentials to every protected resource. To do this without bothering the user, such a client would likely want to store the username and password so that they can be replayed in the future. The protected resources would need to see and verify the user's password on every request, creating an enormous attack surface for sensitive materials.

This grant type, then, can act as a stepping-stone toward a more modern security architecture that uses OAuth's other, more secure grant types. For one, the protected resource no longer needs to know or ever see the user's password, and it can deal only with OAuth tokens. This immediately limits the exposure of the user's credentials across the network and limits the number of components that ever see them. Second, in using this grant type a well-meaning client application no longer needs to store the passwords and transmit them to the resource servers. The client trades them for an access token that it can use at various protected resources. Combined with a refresh token, the user experience is unchanged from before but the security profile is greatly improved over the alternative. Although using something like the authorization code grant type is greatly preferable, using this flow is sometimes better than replaying the user's password to the protected resource on every request.

The way that the grant type works is simple. The client collect's the resource owner's username and password, using whatever interface it has at its disposal, and replays that at the authorization server.

```
POST /token
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3JldC0x

grant_type=password&scope=foo%20bar&username=alice&password=secret
```

The authorization server reads the username and password off the incoming request and compares it with its local user store. If they match, the authorization server issues a token for that resource owner.

If you think this looks a lot like a man-in-the-middle attack, you're not far off. You know that you're not supposed to do this, and why, but we're going to work through how to build it so that you know what not to build in the future, if you can avoid it. Hopefully you'll also be able to see some of the problems inherent in using this grant type by watching how the data is put together. Open up `ch-6-ex-3` and edit the `authorizationServer.js` file to get started. Since this is a back-channel flow, we're going to be working on the token endpoint once again. Look for the code that handles the authorization code grant type.

```
if (req.body.grant_type == 'authorization_code') {
```

We're going to add another clause to this `if` statement, looking for the `password` value in the `grant_type` parameter.

```
} else if (req.body.grant_type == 'password') {
```

Remember that at this point of the code we've already vetted that the client is valid and it has authenticated itself, so now we need to find out who the resource owner is. In our example code, we're storing user information in an in-memory data object called `userInfo`. In a production system, user information, including passwords, is likely stored in a database or directory of some kind. We've provided a simple lookup function that gets the user information object based on the username.

```
var getUser = function(username) {
   return userInfo[username];
};
```

The details of this function don't matter for building OAuth functionality, because a production system will likely be using a database or other user store. We'll use this function to look up the username that was passed in and make sure the user exists, returning an error if it doesn't.

```
var username = req.body.username;
var user = getUser(username);
if (!user) {
   res.status(401).json({error: 'invalid_grant'});
   return;
}
```

Next, we need to see whether the password matches what's stored on our user objects. Since we're storing simple users in memory and the passwords are in plaintext, we're going to do a straight string comparison of the input password. In any sane production system, the password would be hashed and preferably also salted. We'll return an error if the passwords don't match.

```
var password = req.body.password;
if (user.password != password) {
   res.status(401).json({error: 'invalid_grant'});
   return;
}
```

The client can also pass in a `scope` parameter, so we can do the same kinds of scope checks that we did in previous exercises.

```
var rscope = req.body.scope ? req.body.scope.split(' ') : undefined;
var cscope = client.scope ? client.scope.split(' ') : undefined;
if (__.difference(rscope, cscope).length > 0) {
   res.status(401).json({error: 'invalid_scope'});
   return;
}
```

With all of our checks made, we can generate and return the token. Notice that we can (and do) also create a refresh token. By handing the client a refresh token, it doesn't need to store the resource owner's password any longer.

```
var access_token = randomstring.generate();
var refresh_token = randomstring.generate();

nosql.insert({ access_token: access_token, client_id: clientId, scope:
rscope });
nosql.insert({ refresh_token: refresh_token, client_id: clientId, scope:
rscope });

var token_response = { access_token: access_token, token_type: 'Bearer',
refresh_token: refresh_token, scope: rscope.join(' ') };

res.status(200).json(token_response);
```

This generates the regular JSON object that we've come to expect from the token endpoint. The token is functionally identical to one gotten through any other OAuth grant type.

On the client side, we need to first ask the user to type in their username and password. We've set up a form that will prompt the user for their username and password in order to get a token (figure 6.4).

For this exercise, use the username *alice* and the password *password*, from the first user object in the authorization server's `userInfo` collection. If the user enters their information into this form and presses the button, their credentials will be sent via HTTP `POST` to `/username_password` on the client. We'll now set up a listener for that request.

```
app.post('/username_password', function(req, res) {

});
```

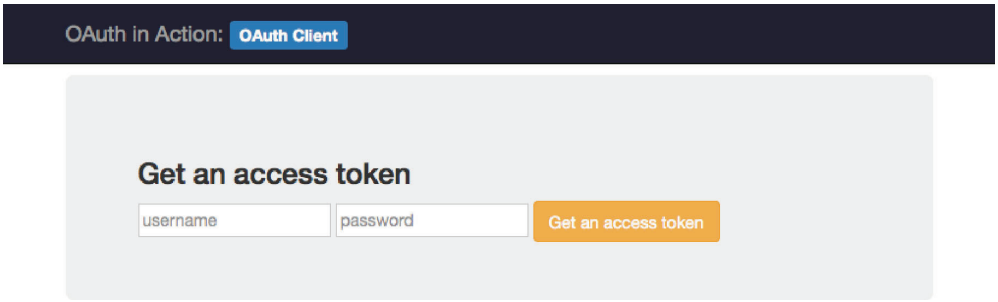

Figure 6.4 The client prompting the user for their username and password

We'll pull the username and password out of the incoming request and pass them along as is to the authorization server, just like a good man-in-the-middle attack. Unlike a real man-in-the-middle attack, we do the right thing and promptly forget about the username and password that we were just told because we're about to get an access token instead.

```
var username = req.body.username;
var password = req.body.password;

var form_data = qs.stringify({
   grant_type: 'password',
   username: username,
   password: password,
   scope: client.scope
});

var headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'Authorization': 'Basic ' + encodeClientCredentials(client.client_id,
   client.client_secret)
};

var tokRes = request('POST', authServer.tokenEndpoint, {
   body: form_data,
   headers: headers
});
```

The response from the authorization server's token endpoint is the same one we've come to expect, so we'll parse out the access token and keep moving with our application as if we hadn't just committed a horrible security faux pas.

```
if (tokRes.statusCode >= 200 && tokRes.statusCode < 300) {
   var body = JSON.parse(tokRes.getBody());

   access_token = body.access_token;

   scope = body.scope;

   res.render('index', {access_token: access_token, refresh_token: refresh_
   token, scope: scope});
```

```
} else {
   res.render('error', {error: 'Unable to fetch access token, server
   response: ' + tokRes.statusCode})
}
```

None of the rest of the client application needs to change. The access token collected here is presented to the protected resource in exactly the same way, shielding the protected resource from knowing that we just saw a user's password in all its plaintext glory. It's important to remember that in the old way of solving this kind of problem, the client would be replaying the user's password directly to the protected resource on every request. Now with this grant type, even if the client isn't doing the absolute best thing here, the protected resource itself now doesn't have to know or see the user's credentials in any fashion.

Now that you know how to use this grant type, if you can at all avoid it, *please don't do it in real life.* This grant type should be used only to bridge clients that would otherwise be dealing with a direct username and password into the OAuth world, and such clients should instead use the authorization code flow in almost all cases as soon as possible. As such, don't use this grant type unless you have no other choice. The internet thanks you.

6.1.4 Assertion grant types

In the first official extension grant types to be published[3] by the OAuth working group, the assertion grant types, the client is given a structured and cryptographically protected item called an *assertion* to give to the authorization server in exchange for a token. You can think of an assertion as something like a certified document such as a diploma or license. You can trust the document's contents to be true as long as you trust the certifying authority's ability to make those statements truthfully (figure 6.5).

Two formats are standardized so far: one using Security Assertion Markup Language (SAML),[4] and another using JSON Web Token (JWT)[5] (which we'll cover in chapter 11). This grant type uses the back channel exclusively, and much like the client credentials flow there may not be an explicit resource owner involved. Unlike the client credentials flow, the rights associated with the resulting token are determined by the assertion being presented and not solely by the client itself. Since the assertion generally comes from a third party external to the client, the client can remain unaware of the nature of the assertion itself.

Like other back-channel flows, the client makes an HTTP POST to the authorization server's token endpoint. The client authenticates itself as usual and includes the assertion as a parameter. The means by which the client can get this assertion vary wildly, and are considered out of scope by many of the associated protocols. The client could be handed the assertion by a user, or by a configuration system, or through another non-OAuth protocol. In the end, as with an access token, it doesn't matter

[3] RFC 7521 https://tools.ietf.org/html/rfc7521
[4] RFC 7522 https://tools.ietf.org/html/rfc7522
[5] RFC 7523 https://tools.ietf.org/html/rfc7523

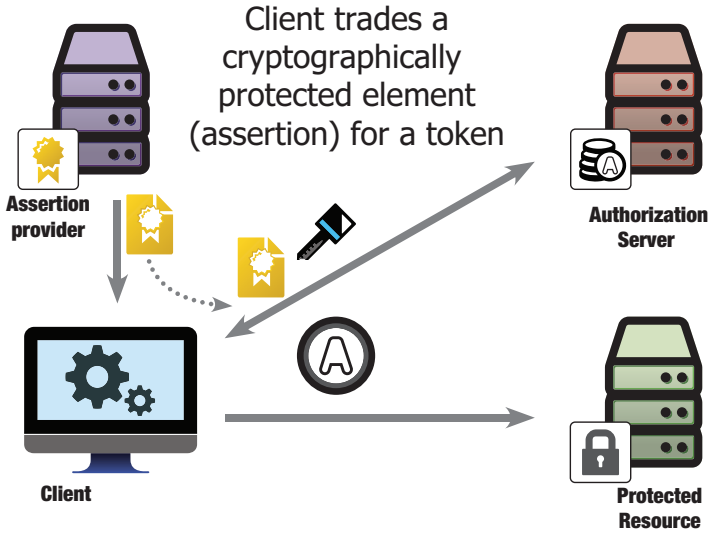

Figure 6.5 The assertion grant type family

how the client got the assertion as long as it's able to present the assertion to the authorization server. In this example, the client is presenting a JWT assertion, which is reflected in the value of the `grant_type` parameter.

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3JldC0x

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6InJzYS0xIn0.eyJpc3MiOi
JodHRwOi8vdHJ1c3QuZXhhbXBsZS5uZXQvIiwic3ViIjoib2F1dGgtY2xpZW50LTEiLCJzY29wZSI
6ImZvbyBiYXIgYmF6IiwiYXVkIjoiaHR0cDovL2F1dGhzZXJ2ZXIuZXhhbXBsZS5uZXQvdG9rZW4i
LCJpYXQiOjE0NjU1ODI5NTYsImV4cCI6MTQ2NTczMzI1NiwianRpIjoiWDQ1cDM1SWZPckRZTmxXO
G9BQ29Xb1djMDQ3V2J3djIifQ.HGCeZh79Va-7meazxJEtm07ZyptdLDu_Ocfw82F1zAT2p6Np6Ia_
vEZTKzGhI3HdqXsUG3uDILBv337VNweWYE7F9ThNgDVD90UYGzZN5VlLf9bzjnB2CDjUWXBhgepSy
aSfKHQhfyjoLnb2uHg2BUb5YDNYk5oqaBT_tyN7k_PSopt1XZyYIAf6-5VTweEcUjdpwrUUXGZ0fl
a8s6RIFNosqt5e6j0CsZ7Eb_zYEhfWXPo0NbRXUIG3KN6DCA-ES6D1TW0Dm2UuJLb-LfzCWsA1W_
sZZz6jxbclnP6c6Pf8upBQIC9EvXqCseoPAykyR48KeW8tcd5ki3_tPtI7vA
```

The body of this example assertion translates to the following:

```
{
  "iss": "http://trust.example.net/",
  "sub": "oauth-client-1",
  "scope": "foo bar baz",
  "aud": "http://authserver.example.net/token",
  "iat": 1465582956,
  "exp": 1465733256,
  "jti": "X45p35IfOrDYNlW8oACoWoWc047Wbwv2"
}
```

The authorization server parsers the assertion, checks its cryptographic protection, and processes its contents to determine what kind of token to generate. This assertion can represent any number of different things, such as a resource owner's identity or a set of allowed scopes. The authorization server will generally have a policy that determines the parties that it will accept assertions from and rules for what those assertions mean. In the end, it generates an access token as with any other response from the token endpoint. The client can then take this token and use it at the protected resource in the normal fashion.

Implementation of this grant type is similar to other back-channel-only flows, wherein the client presents information to the token endpoint and the authorization server issues a token directly. In the real world, you're likely to see assertions used only in limited, usually enterprise, contexts. Generating and processing assertions in a secure manner is an advanced topic worthy of its own set of books, and implementing the assertions flow is left as an exercise to the reader.

6.1.5 Choosing the appropriate grant type

With all of these choices for grant types, it may seem a daunting task to decide which one is the most appropriate for the task at hand. Thankfully, there are a few good ground rules to follow that can guide you in the right direction (figure 6.6).

Is your client acting on behalf of a particular resource owner? And do you have the ability to send that user to a webpage inside their web browser? If so, you'll want to use one

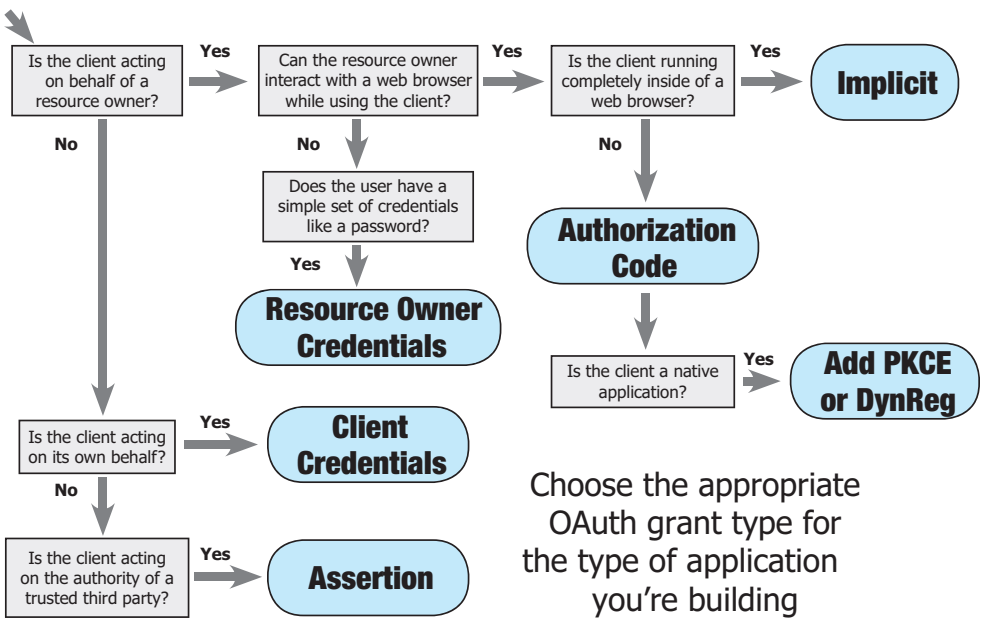

Figure 6.6 Choosing the right grant type

of the redirect-based flows: authorization code or implicit. Which one? That depends on the client.

Is your client completely inside a web browser? This doesn't include applications that execute on a server and whose user interface is accessed through a web browser, only applications that live and die entirely within the browser itself. If so, then you'll want to use the implicit flow, as it is an optimization for this specific case. If not, whether the application is served from a web server or runs natively on the user's computer, you'll want to use the authorization code flow, as it has the best security properties and most flexibility.

Is your client a native application? You should already be using the authorization code grant type, but as we'll see in chapters 7, 10, and 12 you'll want to be using specific security extensions in addition to the authorization code grant type such as dynamic registration (DynReg) or proof key for code exchange (PKCE). We will look at these in more depth as we discuss native applications later in this chapter.

Is your client acting on its own behalf? This includes accessing APIs that don't necessarily map to a single user, such as bulk data transfers. If so, then you should be using the client credentials flow. If you're using an API that has you specify in a parameter which user you're acting on, you should consider using one of the redirect-based flows instead, as that will allow individualized consent and auditing.

Is your client working under the direction of an authoritative third party? Can that third party give you something directly that proves you can act on its behalf? If so, then you should probably be using one of the assertion flows. Which one you use depends on the authorization server and the third party that issues the assertion.

Is your client unable to redirect a user in a browser? Does that user have a simple set of credentials that you can talk them into giving to you? Do you have no other option? If so, then maybe you can use the resource owner credentials flow and be conscientious of its limitations. But don't say we didn't warn you.

6.2 Client deployments

OAuth clients come in many different forms and styles, but they can be broadly categorized into one of three categories: web applications, in-browser applications, and native applications. Each of these has its own strengths and weaknesses, and we'll cover them in turn.

6.2.1 Web applications

The original use case of the OAuth client is the web application. These are applications that execute on a remote server and are accessed through a web browser. The application configuration and its runtime state are held on the web server, and the browser connection is usually connected using a session cookie.

These applications are able to make full use of both front- and back-channel communication methods. Since the user is already interacting through a browser, activating a request on the front channel is as simple as sending an HTTP redirect message

to the browser. Listening for the response on the front channel is equally simple as the application is already listening for HTTP requests. Back channel communication can occur by making an HTTP call directly from the web server running the application. Because of this flexibility, web applications can easily use the authorization code, client credentials, or assertions flows most effectively. Since the fragment component of the request URI isn't usually passed to the server by the browser, the implicit flow doesn't work for web applications in most circumstances.

We've covered several examples and variations of web applications in chapters 2 and 3, so we're not going to cover them in any additional depth here.

6.2.2 Browser applications

Browser applications are those that execute entirely inside the web browser, typically using JavaScript. Although the code for the application does need to be served from a web server, the code itself doesn't execute on the server, and the web server doesn't keep any of the runtime state of the application. Instead, everything about the application happens on the end user's computer inside their web browser.

These clients can easily use the front channel, as sending the user to another page through an HTTP redirect is trivial. Responses from the front channel are also simple, as the client's software does need to be loaded from a web server. However, back-channel communication is more complicated, as browser applications are limited by same-origin policies and other security restrictions designed to prevent cross-domain attacks. Consequently, these types of applications are best suited for the implicit flow, which has been optimized for this case.

Let's take a hands-on look at a browser application. Open up `ch-6-ex-4` and edit `files/client/index.html`. Unlike other examples in this book, we won't be editing the Node.js code this time, but instead we'll be looking at code that's running inside the browser. We still need a client configuration and authorization server configuration in order for this to function, and they've been included as objects at the top of the main function, as in our web application.

```
var client = {
  'client_id': 'oauth-client-1',
  'redirect_uris': ['http://localhost:9000/callback'],
  'scope': 'foo bar'
};

var authServer = {
  authorizationEndpoint: 'http://localhost:9001/authorize'
};

var protectedResource = 'http://localhost:9002/resource';
```

When the user clicks the Authorize button, we generate a front-channel request to the authorization server's authorization endpoint. First, we'll generate a state value and store it in HTML5 local storage so that we can pick it up later.

```
var state = generateState(32);
localStorage.setItem('oauth-state', state);
```

Then we'll build the URI to the authorization endpoint and send the resource owner there with an HTTP redirect.

```
location.href = authServer.authorizationEndpoint + '?' +
   'response_type=token' +
   '&state=' + state +
   '&scope=' + encodeURIComponent(client.scope) +
   '&client_id=' + encodeURIComponent(client.client_id) +
   '&redirect_uri=' + encodeURIComponent(client.redirect_uris[0]);
```

This request is identical to the one used in the web application examples except that the `response_type` has been set to `token`. This application uses a full-page redirect to the authorization server to start this process, which means that the entire application is reloaded and must restart upon callback. An alternative approach is to use an inline frame, or iframe, to send the resource owner to the server.

When the resource owner returns on the redirect URI, we need to be able to listen for that callback and process the response. Our application does this by checking the state of the URI fragment, or hash, when the page is loaded. If the fragment exists, we parse out its components into the access token and scopes.

```
var h = location.hash.substring(1);
var whitelist = ['access_token', 'state']; // for parameters

callbackData = {};

h.split('&').forEach(function (e) {
  var d = e.split('=');

  if (whitelist.indexOf(d[0]) > -1) {
    callbackData[d[0]] = d[1];
  }
});

if (callbackData.state !== localStorage.getItem('oauth-state')) {
  callbackData = null;
  $('.oauth-protected-resource').text("Error state value did not match");
} else {
  $('.oauth-access-token').text(callbackData.access_token);
}
```

From here, our application can start using the access token with protected resources. Note that access to external sites from a JavaScript application still requires cross-domain security configuration, such as CORS, on the part of the protected resource, as we'll discuss in chapter 8. Using OAuth in this type of application allows for a kind of cross-domain session, mediated by the resource owner and embodied by the access token. Access tokens in this case are usually short lived and often limited in scope. To refresh this session, send the resource owner back to the authorization server to get a new access token.

6.2.3 Native applications

Native applications are those that run directly on the end user's device, be it a computer or mobile platform. The software for the application is generally compiled or packaged externally and then installed on the device.

These applications can easily make use of the back channel by making a direct HTTP call outbound to the remote server. Since the user isn't in a web browser, as they are with a web application or a browser client, the front channel is more problematic. To make a front-channel request, the native application needs to be able to reach out to the system web browser or an embedded browser view to get the user to the authorization server directly. To listen for front-channel responses, the native application needs to be able to serve a URI that the browser can be redirected to by the authorization server. This usually takes one of the following forms:

- An embedded web server running on `localhost`
- A remote web server with some type of out-of-band push notification capability to the application
- A custom URI scheme such as `com.oauthinaction.mynativeapp:/` that is registered with the operating system such that the application is called when URIs with that scheme are accessed

For mobile applications, the custom URI scheme is the most common. Native applications are capable of using the authorization code, client credentials, or assertion flows easily, but because they can keep information out of the web browser, it is not recommended that native applications use the implicit flow.

Let's see how to build a native application. Open up `ch-6-ex-5`, and you'll find the code for the authorization server and protected resource in there as usual. However, the client will be in the `native-client` subdirectory this time instead of a `client.js` script in the main folder. All the exercises in this book so far are developed in JavaScript using the Express.js web application framework running on Node.js. A native application doesn't need to be accessible from a web browser, but we still tried to be consistent with the choice of language. For this reason, we chose to use the Apache Cordova[6] platform, which allows us to build a native application using JavaScript.

Do I need to use web technologies to build an OAuth client?

For consistency's sake in all the exercises in this book, we're using many of the same languages and technologies in our native application exercise that we've been using in our web-based applications. However, that's not to say that you *need* to build your own native application using HTML and JavaScript, or any other particular language or platform. An OAuth application needs be able to make direct HTTP ***(continued)***

[6] https://cordova.apache.org/

calls to the back-channel endpoints, launch a system browser for the front-channel endpoints, and listen for the responses from those front-channel endpoints on some kind of URI addressable from the browser. The details of how this happens vary depending on the platform, but these functions are available with many different application frameworks.

Just as before, we tried to focus the attention on OAuth and shield you, the reader, from as many of the platform specific quirks as we could. Apache Cordova is available as a module in Node Package Manager (NPM), so installation is similar to other Node.js modules. Although the details of this vary from system to system, we're going to show an example from a Mac OSX platform.

```
> sudo npm install -g cordova
> npm install ios-sim
```

Now that this is done, let's take a look at the native application's code. Open up `ch-6-ex-5/native-client/` and edit `www/index.html`. As in the browser application exercise, we won't be editing any code this time, but instead we'll be looking at code that's running inside the native application. You run the native application in your computer. In order to do so, you need a few additional steps. Inside the `ch-6-ex-5/native-client/` directory you need to add a runtime platform. Here we're using iOS, and different platforms are available in the Cordova framework.

```
> cordova platform add ios
```

You then need to install a couple of plugins so that your native application can call the system browser and listen to a custom URL scheme.

```
> cordova plugin add cordova-plugin-inappbrowser
> cordova plugin add cordova-plugin-customurlscheme --variable URL_SCHEME=
com.oauthinaction.mynativeapp
```

Finally, we can run our native app.

```
> cordova run ios
```

This should pull up the application in a mobile phone simulator (figure 6.7).

Now let's take a look at the code. The first thing to notice is the client configuration.

```
var client = {
   "client_id": "native-client-1",
   "client_secret": "oauth-native-secret-1",
   "redirect_uris": ["com.oauthinaction.mynativeapp:/"],
   "scope": "foo bar"
};
```

As you can see, the registration details are same as they are for a normal OAuth client. One thing that might catch your attention is the registered `redirect_uris`. This is different from a traditional client because it uses a custom URI scheme, `com.oauthinaction.mynativeapp:/` in this case, rather than a more traditional

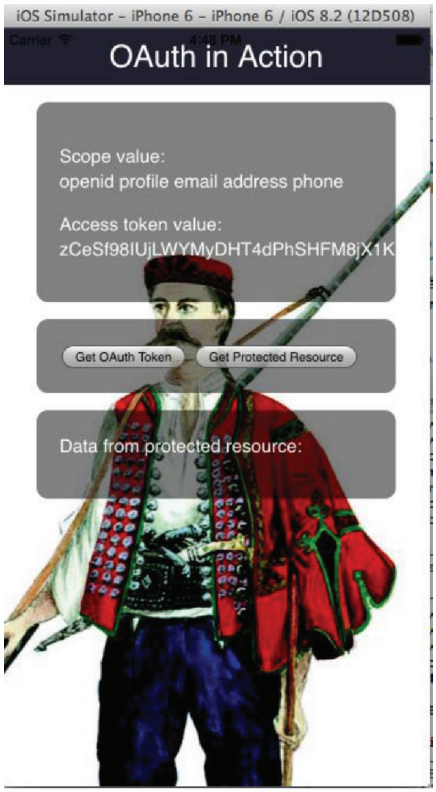

Figure 6.7 A native mobile OAuth client application

`https://`. Whenever the system browser sees a URL starting with `com.oauthinaction.mynativeapp:/`, whether it's from a link clicked by the user or an HTTP redirect from another page or from an explicit launch from another application, our application will get called using a special handler. Inside this handler, we have access to the full URL string that was used in the link or redirect, just as if we were a web server serving the URL through HTTP.

Keeping secrets in native applications

In our exercise, we're using a client secret that's been configured directly into the client as we did with the web application in chapter 3. In a production native application, our exercise's approach doesn't work very well because each copy of the application would have access to the secret, which of course doesn't make it very secret. A few alternative options are available to use in practice. We'll cover this issue in greater detail in section 6.2.4, but for the moment we've opted for consistency between the examples here.

The authorization server and protected resource configuration are the same as for the other examples.

```
var authServer = {
   authorizationEndpoint: 'http://localhost:9001/authorize',
   tokenEndpoint: 'http://localhost:9001/token',
};

var protectedResource = 'http://localhost:9002/resource';
```

Since we're going to use the authorization code flow, when the user clicks the Authorize button, we generate a front-channel request using `response_type=code` request parameter. We still need to generate a state value and store it in our application (using HTML5 local storage in Apache Cordova) so that we can pick it up later.

```
var state = generateState(32);
localStorage.setItem('oauth-state', state);
```

Having done this, we're ready to build the request. This request is identical to the authorization request we used in chapter 3 when we first met the authorization code grant type.

```
var url = authServer.authorizationEndpoint + '?' +
   'response_type=code' +
   '&state=' + state +
   '&scope=' + encodeURIComponent(client.scope) +
   '&client_id=' + encodeURIComponent(client.client_id) +
   '&redirect_uri=' + encodeURIComponent(client.redirect_uris[0]);
```

To initiate the request to the authorization server, we need to invoke the system browser from our application. Since the user isn't already in a web browser, we can't simply use an HTTP redirect as we could with web-based clients.

```
cordova.InAppBrowser.open(url, '_system');
```

After the resource owner authorizes the client, the authorization server redirects them in the system browser to the redirect URI. Our application needs to be able to listen for that callback and process the response as if it were an HTTP server. This is done in the `handleOpenURL` function.

```
function handleOpenURL(url) {
   setTimeout(function() {
        processCallback(url.substr(url.indexOf('?') + 1));
   }, 0);
}
```

This function listens for incoming calls on `com.oauthinaction.mynativeapp:/` and extracts the request parameters from the URI, sending those parameters to the `processCallback` function. In the `processCallback`, we parse out the components to get the `code` and `state` parameters.

```
var whitelist = ['code', 'state']; // for parameters

callbackData = {};
```

```
h.split('&').forEach(function (e) {
var d = e.split('=');

if (whitelist.indexOf(d[0]) > -1) {
   callbackData[d[0]] = d[1];
}
```

We once again need to check that the `state` matches. If it doesn't match, we show an error.

```
if (callbackData.state !== localStorage.getItem('oauth-state')) {
   callbackData = null;
   $('.oauth-protected-resource').text("Error: state value did not match");
```

If the `state` presented is correct, we can trade the authorization code for an access token. We do this by making a direct HTTP call in the back channel. On the Cordova framework, we use the JQuery `ajax` function to make this call.

```
$.ajax({
   url: authServer.tokenEndpoint,
   type: 'POST',
   crossDomain: true,
   dataType: 'json',
   headers: {
        'Content-Type': 'application/x-www-form-urlencoded'
   },
   data: {
        grant_type: 'authorization_code',
        code: callbackData.code,
        client_id: client.client_id,
        client_secret: client.client_secret,
   }
}).done(function(data) {
   $('.oauth-access-token').text(data.access_token);
   callbackData.access_token = data.access_token;
}).fail(function() {
   $('.oauth-protected-resource').text('Error while getting the access
   token');
});
```

Once we have the access token, we can then consume the protected resource API using the given access token. Here we've wired up that call to the event handler for our button.

```
function handleFetchResourceClick(ev) {
   if (callbackData != null ) {
   $.ajax({
        url: protectedResource,
        type: 'POST',
        crossDomain: true,
        dataType: 'json',
        headers: {
                'Authorization': 'Bearer ' + callbackData.access_token
        }
   }).done(function(data) {
        $('.oauth-protected-resource').text(JSON.stringify(data));
```

```
    }).fail(function() {
         $('.oauth-protected-resource').text('Error while fetching the
    protected resource');
    });
}
```

The native application can now use this token for as long as it needs to access the protected resource. Since we used the authorization code flow, we can also be issued a refresh token to be used when the access token expires. This approach allows us to have a smooth user experience in our native application while keeping the security profile of OAuth.

6.2.4 Handling secrets

The purpose of the client secret is to let an instance of client software authenticate itself to the authorization server, apart from any authorizations conferred to it by the resource owner. The client secret isn't available to the resource owner or the browser, allowing it to uniquely identify the client software application. In OAuth 1.0, every client was expected to have its own client secret (known as a *consumer key* in that specification), regardless of the kind of client that it was. However, as we've seen throughout this chapter, not all OAuth clients are created equal. Although a web application can be configured with a client secret away from the browser and end user, native applications and browser applications can't.

The problem comes from needing to differentiate between *configuration time secrets*, which every copy of a client gets, and *runtime secrets*, which are distinct for each instance. Client secrets are configuration time secrets because they represent the client software itself and are configured into the client software. Access tokens, refresh tokens, and authorization codes are all runtime secrets because they're stored by the client software after it is deployed and running. Runtime secrets do still need to be stored securely and protected appropriately, but they're designed to be easily revocable and rotatable. Configuration time secrets, in contrast, are generally things that aren't expected to change often.

In OAuth 2.0, this dichotomy is addressed by removing the requirement for all clients to have a client secret and instead defining two classes of clients, *public clients* and *confidential clients*, based on their ability to keep a configuration time secret.

Public clients, as the name suggests, are unable to hold configuration time secrets and therefore have no client secret. This is usually because the code for the client is exposed to the end user in some fashion, either by being downloaded and executed in a browser or by executing natively on the user's device. Consequently, most browser applications and many native applications are public clients. In either case, each copy of the client software is identical and there are potentially many instances of it. The user of any instance could extract the configuration information for that instance, including any configured client ID and client secret. Although all instances share the same client ID, this doesn't cause a problem because the client ID isn't intended to be a secret value. Anyone attempting to impersonate this client by copying its client

ID will still need to use its redirect URIs and be bound by other measures. Having an additional client secret, in this case, does no good because it could be extracted and copied along with the client ID.

A potential mitigation is available for applications that use the authorization code flow in the form of Proof Key for Code Exchange (PKCE), discussed in chapter 10. The PKCE protocol extension allows a client to more tightly bind its initial request to the authorization code that it receives, but without using a client secret or equivalent.

Confidential clients are able to hold configuration time secrets. Each instance of the client software has a distinct configuration, including its client ID and secret, and these values are difficult to extract by end users. A web application is the most common type of confidential client, as it represents a single instance running on a web server that can handle multiple resource owners with a single OAuth client. The client ID can be gathered as it is exposed through the web browser, but the client secret is passed only in the back channel and is never directly exposed.

An alternative approach to this problem is to use dynamic client registration, discussed in depth in chapter 12. By using dynamic client registration, an instance of a piece of client software can register itself at runtime. This effectively turns what would otherwise need to be a configuration time secret into a runtime secret, allowing a higher level of security and functionality to clients that would otherwise be unable to use it.

6.3 Summary

OAuth 2.0 gives you a lot of options inside a common protocol framework.

- The canonical authorization code grant type can be optimized in several ways for different deployments.
- Implicit grants can be used for in-browser applications without a separate client.
- Client credentials grants and assertion grants can be used for server-side applications without an explicit resource owner.
- Nobody should be using the resource owner credentials grant unless they really have no other choice.
- Web applications, browser applications, and native applications all have different quirks for using OAuth, but all share a common core.
- Confidential clients can keep client secrets, but public clients can't.

Now that we've had a thorough look at how things are supposed to work in an OAuth ecosystem, we're going to take a look at some of the things that can go wrong. Read on to learn how to deal with vulnerabilities found in implementations and deployments of OAuth.